Independent Choice Logic Interpreter
Version 0.2.1
PROLOG CODE*

David Poole
Department of Computer Science,
University of British Columbia,
2366 Main Mall,
Vancouver, B.C. Canada V6T 1Z4
Phone: (604) 822-6254
Fax: (604) 822-5485
Email: poole@cs.ubc.ca
URL: http://www.cs.ubc.ca/spider/poole

June 11, 1998

Abstract

This paper gives the code for a simple independent choice logic [1, 2] interpreter. This is based on a naive Prolog search (rather than some best first, or iterative deepening search).

It includes negation as failure, controllables (although is a very limited form — they are not chosen, but the user can control them), and debugging facilities (including tracing, traversing proof trees, and automatic detection of non-disjoint rules).

This is experimental code. It is written to let us explore with ideas. I make no warranty as to its suitability for anything. Use at your own risk.

1 Syntax

The following commands can be used in a file or as a user command to the prolog prompt. Note that all commands end with a period.

`rule`(R)**`.`** The facts are given in the form of `rule(`R`)`, where R is either a rule of the form `H :- B` or is just an atom (similar to the use of $clause$ in many Prolog systems). B is a body made up of `true`, atoms, conjunctions (using "`&`"), disjunctions (using "`;`") and negations (using "`~`"). This code assumes that rules are disjoint.

H **`<-`** B**`.`** is the same as `rule((H :- B))`. Rules with empty bodies (facts) must be written as `H <- true.` or as `rule(H).`

`random`$([h_1 : p_1, \cdots, h_n : p_n])$**`.`** declares the h_i to be pairwise disjoint hypotheses, with $P(h_i) = p_i$.

`random`$(X, h, [x_1 : p_1, \cdots, x_n : p_n])$**`.`** where h is an atom that contains variable X, and the x_1 are different terms, declares the atoms h with X replaced by each x_i to be pairwise disjoint hypotheses with $P(h[x_i]) = p_i$.

`controllable`$([h_1, \cdots, h_n])$**`.`** declares the h_i to be pairwise disjoint controllable variables (i.e., the agent can choose one of the h_i.)

The following commands can be used as user commands to the prolog prompt:

`explain`(G, C)**`.`** asks to find all explanations of G given controllables C.

`thcons`(***`filename`***)**`.`** loads a file called *filename*. This does not erase any definitions in the database.

`tracing(F).` sets tracing to have status F which is one of {`yes,no,duals`}. `duals` traces only the duals (i.e., the explanations of the negation of an atom).

`debug(F).` sets debugging to have status F which is one of {`yes,no`}. This lets you choose which rules get selected, so you can pinpoint missing cluases (i.e., when an answer wasn't returned).

`help.` gives a list of commands.

`how`(G, C, N)**`.`** is used to explain the Nth explanation of G given C. Called after `explain`(G, C)`.`

`diff`(G, C, N, M)**`.`** prints the differences in the proof tree for the Nth and Mth explanation of G given C. Called after `explain`(G, C).

check(G)**.** checks for disjoint rules in the explanations of G. Called after `explain`(G, C)**..**

check_disj(G_1, G_2)**.** checks for cases in which G_1 and G_2 are both true. Called after `explain`(G_1, C) and `explain`(G_2, C).

recap(G). recaps the explanations of G, with posterior probabilities of each explanation (i.e., given G). Called after `explain`(G, C).

recap. gives the prior probabilities of every goal explained.

check_undef. checks for undefined atoms — those atoms in the body of rules for which there is no corresponding definition.

clear. clears the knowledge base. This should be done before reloading clauses.

2 Code

2.1 Operators

The "if" operator is written as "`<-`". In bodies, conjunction is represented as "`&`", disjunction as "**;**", negation as failure as "`~`", and inequality as "`\=`.

```
 */
:- op(1060, xfy, [ & ]).
:- op(900,fy, ~).
:- op(700,xfx, \=).
:- op(1150, xfx, <- ).
/*
```

The following declare the predicates that are used to store the object-level knowledge base.

```
 */
:- dynamic rul/2.
:- dynamic control/1.
:- dynamic hypothesis/2.
:- dynamic nogood/2.
/*
```

2.2 Clearing the knowledge base

```
 */
clear :-
   retractall(rul(_,_)),
   retractall(control(_)),
   retractall(hypothesis(_,_)),
   retractall(nogood(_,_)),
   retractall(done(_,_,_,_,_)).
/*
```

2.3 Declaring Rules

$rule(R)$ where R is the form of a Prolog rule. This asserts the rule produced.

`h <- b` is the same as $rule(h,b)$.

```
 */
(H <- B) :- rule((H :- B)).
rule((H :- B)) :- !,
   assertz(rul(H,B)).
rule(H) :-
   assertz(rul(H,true)).
/*
```

$lemma(G)$ is here to make the program upwardly compatible with the version that includes lemmata. The declaration is ignored here.

```
 */
lemma(_).
/*
```

2.4 Declaring Hypotheses

`random`$([h_1 : p_1, \cdots, h_n : p_n])$.

declares the h_i to be pairwise disjoint hypotheses, with $P(h_i) = p_i$. It should be the case that

$$\sum_{i=1}^{n} p_i = 1$$

This asserts $hypothesis(h_i, p_i)$ for each i and asserts $ngood(h_i, h_j)$ for each $i \neq j$.

```
 */
:- op( 500, xfx, : ).
:- dynamic hypothesis/2.
random(L) :-
   probsum(L,T),
   randomt(L,T).
probsum([],0).
probsum([_:P|R],P1) :-
   probsum(R,P0),
   P1 is P0+P.
randomt([],_).
randomt([H:P|R],T) :-
   NP is P/T,
   assertz(hypothesis(H,NP)),
   make_hyp_disjoint(H,R),
   randomt(R,T).
/*
```

```
 */
make_hyp_disjoint(_,[]).
make_hyp_disjoint(H,[H2 : _ | R]) :-
   asserta(nogood(H,H2)),
   asserta(nogood(H2,H)),
   make_hyp_disjoint(H,R).
/*
```

$$\texttt{random}(X, h, [x_1 : p_1, \cdots, x_n : p_n]).$$

where X is a variable and h is an atom that contains X free, and the x_1 are different terms, is an abbreviation for

$$\texttt{random}([h[X \leftarrow x_1] : p_1, \cdots, h[X \leftarrow x_n] : p_n]).$$

Where $h[X \leftarrow x_1]$ is the atom h with X replaced by x_i.

```
 */
random(X,H,L) :-
   repvar(X,X1,H,H1),
   asserta((nogood(H,H1) :- dif(X,X1))),
   probsum(L,T),
   random_each(X,H,L,T).
```

```
random_each(_,_,[],_).
random_each(X,H,[X:P|_],T) :-
   NP is P/T,
   asserta(hypothesis(H,NP)),
   fail.
random_each(X,H,[_|R],T) :-
   random_each(X,H,R,T).
/*
```

`controllable`$([h_1, \cdots, h_n])$.

declares the h_i to be pairwise disjoint controllable hypotheses.

This asserts $control(h_i)$ for each i and asserts $ngood(h_i, h_j)$ for each $i \neq j$.

```
 */
:- op(  500, xfx,  : ).
controllable([]).
controllable([H|R]) :-
   asserta(control(H)),
   make_cont_disjoint(H,R),
   controllable(R).
/*


 */
make_cont_disjoint(_,[]).
make_cont_disjoint(H,[H2 | R]) :-
   asserta(nogood(H,H2)),
   asserta(nogood(H2,H)),
   make_cont_disjoint(H,R).
/*
```

3 The Internals of the Interpreter

3.1 Meta-interpreter

The meta-interpreter is implemented using the relation:

$$prove(G, C0, C1, R0, R1, P0, P1, T)$$

where

G is the goal to be proved.

$R1 - R0$ is a difference list of random assumptions to prove G.

$C1 - C0$ is a difference list of controllable assumptions to prove G.

$P0$ is the probability of $R0$, $P1$ is the probability of $R1$.

T is the returned proof tree.

The first rules defining $prove$ are the special purpose rules for commands that are defined in the system.

```
 */
prove(ans(A),C,C,R,R,P,P,ans(A)) :- !,
   ans(A,C,R,P).
prove(report_cp,C,C,R,R,P,P,_) :- !,
   report_cp(C,R,P).
prove(report_evidence,C,C,R,R,P,P,_) :- !,
   report_evidence(C,R,P).
/*
```

The remaining rules are the real definition

```
 */
prove(true,C,C,R,R,P,P,true) :- !.
prove((A & B),C0,C2,R0,R2,P0,P2,(AT & BT)) :- !,
   prove(A,C0,C1,R0,R1,P0,P1,AT),
   prove(B,C1,C2,R1,R2,P1,P2,BT).
prove((A ; _),C0,C2,R0,R2,P0,P2,AT) :-
   prove(A,C0,C2,R0,R2,P0,P2,AT).
prove((_ ; B),C0,C2,R0,R2,P0,P2,BT) :-
   prove(B,C0,C2,R0,R2,P0,P2,BT).
prove((~ G),C0,C0,R0,R2,P0,P3,if(G,not)) :-
   findall(R2,prove(G,C0,_,R0,R2,P0,_,_), ExpG),
   duals(ExpG,R0,[exp(R0,P0)],ADs),
   make_disjoint(ADs,MDs),
   ( (tracn(yes); tracn(duals)) ->
        writeln(['   Proved ~ ',G ,', assuming ',R0,'.']) ,
        writeln(['     explanations of ',G,': ',ExpG]),
        writeln(['     duals: ',ADs]),
        writeln(['     disjointed duals: ',MDs])
     ; true),!,
```

```
   member(exp(R2,P3),MDs).
prove(H,_,_,R,_,P,_,_) :-
   tracn(yes),
   writeln(['Proving: ',H,' assuming: ',R,' prob=',P]),
   fail.
prove(H,C,C,R,R,P,P,if(H,assumed)) :-
   hypothesis(H,_),
   member(H,R),
   ( tracn(yes)
     -> writeln(['   Already assumed: ',H ,'.'])
    ; true).
prove(H,C,C,R,[H|R],P0,P1,if(H,assumed)) :-
   hypothesis(H,PH),
   \+ member(H,R),
   PH > 0,
   good(H,R),
   P1 is P0*PH,
   ( tracn(yes) -> writeln(['   Assuming: ',H ,'.']) ; true).
prove(H,C,C,R,R,P,P,if(H,given)) :-
   control(H),member(H,C),!,
   ( tracn(yes) -> writeln(['   Given: ',H,'.']) ; true).
prove(H,C,C,R,R,P,P,if(H,builtin)) :-
   builtin(H), call(H).
prove(A \= B,C,C,R,R,P,P,if(A \= B,builtin)) :-
   dif(A,B).
prove(G,C0,C1,R0,R1,P0,P1,if(G,BT)) :-
   rul(G,B),
   ( tracn(yes) ->
     writeln(['   Using rule: ',G ,' <- ',B,'.'])
   ; true),
   ( debgn(yes) -> deb(G,B) ; true),
   tprove(G,B,C0,C1,R0,R1,P0,P1,BT),
   ( tracn(yes) ->
     writeln(['   Proved: ',G ,' assuming ',R1,'.'])
     ; true).
prove(H,_,_,R,_,P,_,_) :-
   tracn(yes),
   writeln(['Failed: ',H,' assuming: ',R,' prob=',P]),
   fail.
tprove(_,B,C0,C1,R0,R1,P0,P1,BT) :-
```

```
   prove(B,C0,C1,R0,R1,P0,P1,BT).
tprove(G,_,_,_,R,_,P,_,_) :-
   tracn(yes),
   writeln([' Retrying: ',G,' assuming: ',R,' prob=',P]),
   fail.
/*
```

We allow many built in relations to be evaluated directly by Prolog.

```
 */
:- dynamic builtin/1.
%:- multifile builtin/1.
builtin((_ is _)).
builtin((_ < _)).
builtin((_ > _)).
builtin((_ =< _)).
builtin((_ >= _)).
builtin((_ = _)).
/*

 */
deb(G,B) :-
   writeln([' Use rule: ',G ,' <- ',B,'? [y, n or off]']),
   read(A),
  ( A = y -> true ;
    A = n -> fail ;
    A = off -> debug(off) ;
    true -> writeln(['y= use this rule, n= use another rule, off=debugging off']) ,
       deb(G,B)
   ).

/*
```

3.2 Negation

$duals(Es, R0, D0, D1)$

is true if Es is a list of composite choices (all of whose tail is $R0$), and $D1 - D2$ is a list of $exp(R1, P1)$ such that $R1 - R0$ is a hitting set of negations of Es.

```
 */
duals([],_,D,D).
```

```
duals([S|L],R0,D0,D2) :-
   split_each(S,R0,D0,[],D1),
   duals(L,R0,D1,D2).
/*
```

$$split\_each(S, R0, D0, D, D1)$$

is true if S is a composite choice (with tail $R0$), and $D2$ is D together with the hitting set of negations of $D0$.

```
 */
split_each(R0,R0,_,D0,D0) :- !.
split_each([A|R],R0,D0,PDs,D2) :-
   negs(A,NA),
   add_to_each(A,NA,D0,PDs,D1),
   split_each(R,R0,D0,D1,D2).
/*
```

$$add\_to\_each(S, R0, D0, D, D1)$$

is true if S is a composite choice (with tail $R0$), and $D2$ is D together with the hitting set of negations of $D0$.

```
 */
add_to_each(_,_,[],D,D).
add_to_each(A,NA,[exp(E,_)|T],D0,D1) :-
   member(A,E),!,
   add_to_each(A,NA,T,D0,D1).
add_to_each(A,NA,[exp(E,PE)|T],D0,D2) :-
   bad(A,E),!,
   insert_exp(exp(E,PE),D0,D1),
   add_to_each(A,NA,T,D1,D2).
add_to_each(A,NA,[B|T],D0,D2) :-
   ins_negs(NA,B,D0,D1),
   add_to_each(A,NA,T,D1,D2).
/*
```

$$ins\_negs(NA, B, D0, D1)$$

is true if adding the elements of NA to composite choice B, and adding these to $D0$ produces $D1$.

```
 */
ins_negs([],_,D0,D0).
ins_negs([N|NA],exp(E,PE),D,D2) :-
   hypothesis(N,PN),
   P is PN * PE,
   insert_exp(exp([N|E],P),D,D1),
   ins_negs(NA,exp(E,PE),D1,D2).
/*
```

$$insert\_exp(E, L0, L1)$$

is true if inserting composite choice E into list $L0$ produces list $L1$. Subsumed elements are removed.

```
 */
insert_exp(exp(_,0.0),L,L) :-!.
insert_exp(E,[],[E]) :- !.
insert_exp(exp(E,_),D,D) :-
   member(exp(E1,_),D),
   subset(E1,E),!.
insert_exp(exp(E,P),[exp(E1,_)|D0],D1) :-
   subset(E,E1),!,
   insert_exp(exp(E,P),D0,D1).
insert_exp(exp(E,P),[E1|D0],[E1|D1]) :-
   insert_exp(exp(E,P),D0,D1).
/*
```

3.3 Making Composite Choices Disjoint

$$make\_disjoint(L, SL)$$

is true if L and SL are lists of the form $exp(R, P)$, such that $L1$ is a subset of L containing minimal elements with minimal R-values.

```
 */
make_disjoint([],[]).
make_disjoint([exp(R,P)|L],L2) :-
   member(exp(R1,_),L),
   \+ incompatible(R,R1),!,
   member(E,R1), \+ member(E,R),!,
   negs(E,NE),
```

```
   split(exp(R,P),NE,E,L,L1),
   make_disjoint(L1,L2).
make_disjoint([E|L1],[E|L2]) :-
   make_disjoint(L1,L2).
split(exp(R,P),[],E,L,L1) :-
   hypothesis(E,PE),
   P1 is P*PE,
   insert_exp1(exp([E|R],P1),L,L1).
split(exp(R,P),[E1|LE],E,L,L2) :-
   hypothesis(E1,PE),
   P1 is P*PE,
   split(exp(R,P),LE,E,L,L1),
   insert_exp1(exp([E1|R],P1),L1,L2).
negs(E,NE) :-
   findall(N,nogood(E,N),NE).
insert_exp1(exp(_,0.0),L,L) :-!.
insert_exp1(exp(E,_),D,D) :-
   member(exp(E1,_),D),
   subset(E1,E),!.
insert_exp1(exp(E,P),D,[exp(E,P)|D]).
/*
```

3.4 Nogoods

We assume three relations for handling $nogoods$:

$$good(A, L)$$

fails if $[A|L]$ has a subset that has been declared nogood. We can assume that no subset of L is nogood (this allows us to more efficiently index nogoods).

$$allgood(L)$$

fails if L has a subset that has been declared nogood.

```
 */
allgood([]).
allgood([H|T]) :-
   good(H,T),
   allgood(T).

good(A,T) :-
```

```
   \+ ( makeground((A,T)), bad(A,T)).
bad(A,[B|_]) :-
   nogood(A,B).
bad(A,[_|R]) :-
   bad(A,R).
/*
```

3.5 Explaining

To find an explanation for a subgoal G given controllables C and building on random assumables R, we do an $explain(G, C, R)$. Both R and C are optional.

```
 */
:- dynamic done/4.
explain(G) :-
   explain(G,[],[]).
explain(G,C) :-
   explain(G,C,[]).
explain(G,C,R) :-
   statistics(runtime,_),
   ex(G,C,R).
:- dynamic false/6.
/*
```

$ex(G, C, R)$ tries to prove G with controllables C and random assumptions R. It repeatedly proves G, calling ans for each successful proof.

```
 */
:- dynamic done/5.
ex(G,C,R0) :-
   prove(G,C,_,R0,R,1,P,T),
   ans(G,C,R0,R,P,T), fail.
ex(G,C,R) :-
   done(G,C,R,_,Pr),
   append(C,R,CR),
   nl,
   writeln(['Prob( ',G,' | ',CR,' ) = ',Pr]),
   statistics(runtime,[_,Time]),
   writeln(['Runtime: ',Time,' msec.']).
ex(G,C,R) :-
   \+ done(G,C,R,_,_),
```

```
   append(C,R,CR),
   nl,
   writeln(['Prob( ',G,' | ',CR,' ) = ',0.0]),
   statistics(runtime,[_,Time]),
   writeln(['Runtime: ',Time,' msec.']).
ans(G,C,R0,R,P,T) :-
   allgood(R),
   ( retract(done(G,C,R0,Done,DC))
     -> true
     ; Done=[], DC=0),
   DC1 is DC+P,
   asserta(done(G,C,R0,[expl(R,P,T)|Done],DC1)),
   nl,
   length(Done,L),
   append(C,R0,Given),
   writeln(['***** Explanation ',L,' of ',G,' given ',Given,':']),
   writeln([R]),
   writeln(['Prior = ',P]).
/*
```

recap is used to give a list of all conditional probabilities computed.

```
 */
recap :-
   done(G,C,R,_,Pr),
   append(C,R,CR),
   writeln(['Prob( ',G,' | ',CR,' ) = ',Pr]),
   fail.
recap.
recap(G) :-
   recap(G,_,_).
recap(G,C) :-
   recap(G,C,_).
recap(G,C,R) :-
   done(G,C,R,Expls,Pr),
   append(C,R,CR),
   writeln(['Prob( ',G,' | ',CR,' ) = ',Pr]),
   writeln(['Explanations:']),
   recap_each(_,Expls,Pr).
recap_each(0,[],_).
recap_each(N,[expl(R,P,_)|L],Pr) :-
```

```
   recap_each(N0,L,Pr),
   N is N0+1,
   PP is P/Pr,
   writeln([N0,': ',R,' Post Prob=',PP]).

/*
```

4 Debugging

4.1 Help

```
 */
h <- user_help.
user_help :- writeln([
'rule(R).','
  asserts either a rule of the form H :- B or an atom.','
H <- B. ','
   is the same as rule((H :- B)). ','
   Rules with empty bodies (facts) must be written as H <- true. or as rule(H).','
random([h1:p1,...,hn:pn]).','
   declares the hi to be pairwise disjoint hypotheses, with P(hi)=pi. ','
random(X,h,[x1:p1,...,xn:pn]).','
   declares h[X/xi] to be pairwise disjoint hypotheses with P(h[X/xi])=pi.','
controllable([h1,...,hn]).','
   declares the hi to be pairwise disjoint controllable variables.','
explain(G,C). ','
   finds explanations of G given list of controlling values C.','
how(G,C,R,N).','
   is used to explain the Nth explanation of G given controllables C,','
   and randoms R.','
diff(G,C,N,M) ','
   prints difference in the proof tree for the Nth and Mth explanation','
   of G given C.','
check(G,C).','
   checks for disjoint rules in the explanations of G given C.','
recap(G). ','
   recaps the explanations of G, with posterior probabilities (given G).','
recap. ','
   gives the prior probabilities of everything explained.','
thcons(filename). ','
```

```
   loads a file called filename. ','
tracing(F). ','
    sets tracing to have status F which is one of {yes,no,duals}.','
debug(F). ','
    sets debugging to have status F which is one of {yes,no}.','
check_undef.','
    checks for undefined atoms in the body of rules.','
clear.','
    clears the knowedge base. Do this before reloading.','
    Reconsulting a program will not remove old clauses and declarations.','
help.','
    print this message.']).
/*
```

4.2 Tracing

Tracing is used to trace the details of the search tree. It is ugly except for very small programs.

```
 */
:- dynamic tracn/1.
tracing(V) :-
   member(V,[yes,no,duals]),!,
   retractall(tracn(_)),
   asserta(tracn(V)).
tracing(V) :-
   member(V,[on,y]),!,
   retractall(tracn(_)),
   asserta(tracn(yes)).
tracing(V) :-
   member(V,[off,n]),!,
   retractall(tracn(_)),
   asserta(tracn(no)).
tracing(_) :-
   writeln(['Argument to tracing should be in {yes,no,duals}.']),
   !,
   fail.
tracn(no).
user_help :- unix(shell('more help')).
/*
```

4.3 Debugging

Debugging is useful for determining why a program failed.

```
 */
:- dynamic debgn/1.
debgn(no).

debug(V) :-
   member(V,[yes,on,y]),!,
   retractall(debgn(_)),
   assert(debgn(yes)).
debug(V) :-
   member(V,[no,off,n]),!,
   retractall(debgn(_)),
   assert(debgn(no)).
debug(_) :-
   writeln(['Argument to debug should be in {yes,no}.']), !, fail.
/*
```

4.4 How was a goal proved?

The programs in this section are used to explore how proofs were generated.

$$how(G, C, R, N)$$

is used to explain the Nth explanation of G given controllables C, and randoms R.
R and C are optional.

```
 */
how(G,N) :-
   how(G,[],[],N).
how(G,C,N) :-
   how(G,C,[],N).
how(G,C,R,N) :-
   tree(G,C,R,N,T),
   traverse(T).
/*
```

$$tree(G, C, R, N, NT)$$

is true if NT is the proof tree for the Nth explanation of G given $C \wedge R$.

```
 */
tree(G,C,R,N,NT) :-
   done(G,C,R,Done,_),
   nthT(Done,N,NT).

nthT([expl(_,_,T) |R],N,T) :-
   length(R,N),!.
nthT([_|R],N,T) :-
   nthT(R,N,T).
/*
```

$$traverse(T)$$

is true if T is a tree being traversed.

```
 */
traverse(if(H,true)) :-
    writeln([H,' is a fact']).
traverse(if(H,builtin)) :-
    writeln([H,' is built-in.']).
traverse(if(H,assumed)) :-
    writeln([H,' is assumed.']).
traverse(if(H,given)) :-
    writeln([H,' is a given controllable.']).
traverse(if(H,not)) :-
    writeln([~ H,' is a negation - I cannot trace it. Sorry.']).
traverse(if(H,B)) :-
    B \== true,
    B \== builtin,
    B \== assumed,
    B \== given,
    writeln([H,' :-']),
    printbody(B,1,Max),
    read(Comm),
    interpretcommand(Comm,B,Max,if(H,B)).
/*
```

$$printbody(B, N)$$

is true if B is a body to be printed and N is the count of atoms before B was called (this assumes that “&” is left-associative).

```
 */
printbody((A&B),N,N2) :-
   printbody(A,N,N),
   N1 is N+1,
   printbody(B,N1,N2).
printbody(if(H,not),N,N) :-!,
   writeln(['    ',N,': ~ ',H]).
printbody(if(H,_),N,N) :-
   writeln(['    ',N,': ',H]).
printbody(true,N,N):-!,
   writeln(['    ',N,': true ']).
printbody(builtin,N,N):-!,
   writeln(['    ',N,': built in ']).
printbody(assumed,N,N):-!,
   writeln(['    ',N,': assumed ']).
printbody(given,N,N):-!,
   writeln(['    ',N,': given ']).
/*
```

$$interpretcommand(Comm, B)$$

interprets the command $Comm$ on body B.

```
 */
interpretcommand(N,B,Max,G) :-
   integer(N),
   N > 0,
   N =< Max,
   nth(B,N,E),
   traverse(E),
   traverse(G).
interpretcommand(up,_,_,_).
interpretcommand(N,_,Max,G) :-
   integer(N),
   (N < 1 ; N > Max),
   writeln(['Number out of range: ',N]),
   traverse(G).
interpretcommand(help,_,_,G) :-
   writeln(['Give either a number, up or exit. End command with a Period.']),
   traverse(G).
interpretcommand(C,_,_,G) :-
```

```
   \+ integer(C),
   C \== up,
   C \== help,
   C \== exit,
   C \== end_of_file,
   writeln(['Illegal Command: ',C,'   Type "help." for help.']),
   traverse(G).

% nth(S,N,E) is true if E is the N-th element of conjunction S
nth(A,1,A) :-
   \+ (A = (_&_)).
nth((A&_),1,A).
nth((_&B),N,E) :-
   N>1,
   N1 is N-1,
   nth(B,N1,E).
/*
```

4.5 Diff

$$diff(G,C,N,M)$$

prints the differences in the proof tree for the Nth and Mth explanation of G given C.

```
 */
diff(G,C,N,M) :-
   tree(G,C,N,NT),
   tree(G,C,M,MT),
   diffT(NT,MT).
/*
```

$$diffT(T1,T2)$$

prints the differences in the proof trees $T1$ and $T2$.

```
 */
diffT(T,T) :-
   writeln(['Trees are identical']).

diffT(if(H,B1),if(H,B2)) :-
```

```
   immdiff(B1,B2),!,
   writeln([H,' :-']),
   printbody(B1,1,N1),
   writeln([H,' :-']),
   printbody(B2,N1,_).
diffT(if(H,B1),if(H,B2)) :-
   diffT(B1,B2).
diffT((X&Y),(X&Z)) :- !,
   diffT(Y,Z).
diffT((X&_),(Y&_)) :-
   diffT(X,Y).

immdiff((A&_),(B&_)) :-
   immdiff(A,B).
immdiff((_&A),(_&B)) :-
   immdiff(A,B).
immdiff((_&_),if(_,_)).
immdiff(if(_,_),(_&_)).
immdiff(if(A,_),if(B,_)) :-
   \+ A = B.
immdiff(if(_,_),B) :-
   atomic(B).
immdiff(A,if(_,_)) :-
   atomic(A).
/*
```

4.6 Check

$$check(G, C, R)$$

checks the explanations of G given controllables C and randoms R for rules which violate the disjoint assumptions assumption. The two rules which are not disjoint are returned.

```
 */
check :-
   check(_,_,_).
check(G) :-
   check(G,_,_).
check(G,C) :-
```

```
   check(G,C,_).
check(G,C,R) :-
   done(G,C,R,Done,_),
   check_done(Done).

check_done([expl(R1,_,T1)|D]) :-
   memberR(expl(R2,_,T2),D,DR),
   \+ incompatible(R1,R2),
   length(D,LD),
   length(DR,L2),
   union(R1,R2,R),
   writeln(['Non-disjoint rules ',LD,' & ',L2,' assuming ',R]),
   diffT(T1,T2).
check_done([_|D]) :-
   check_done(D).
/*
```

4.7 Check Disjoint Explanations

$$check\_disj(G0, G1)$$

checks whether explanations of $G0$ and $G1$ are disjoint. This is useful when $G0$ and $G1$ should be incompatible.

```
 */
check_disj(G0,G1):-
   check_disj(G0,G1,_,_).
check_disj(G0,G1,C,R):-
   done(G0,C,R,D0,_),
   done(G1,C,R,D1,_),
   memberR(expl(R0,_,_),D0,LD0),
   memberR(expl(R1,_,_),D1,LD1),
   \+ incompatible(R0,R1),
   length(LD0,L0),
   length(LD1,L1),
   append(C,R,CR),
   writeln(['Explanation ',L0,' of ',G0,' and ',L1,' of ',G1,
       ', given ',CR,' are compatible assuming:']),
   union(R0,R1,R),
   writeln([R]).
incompatible(R1,R2) :-
```

```
   member(A1,R1),
   member(A2,R2),
   nogood(A1,A2).
/*
```

4.8 Checking for undefined atoms

$check_undef$ searches through the knowledge base looking for a rule containing an atom in the body which doesn't have a corresponding definition (i.e., a clause with it at the head, or an atomic choice).

```
 */
check_undef :-
   rul(H,B),
   body_elt_undefined(B,H,B).
check_undef.

body_elt_undefined(true,_,_) :- !,fail.
body_elt_undefined((A&_),H,B) :-
   body_elt_undefined(A,H,B).
body_elt_undefined((_&A),H,B) :- !,
   body_elt_undefined(A,H,B).
body_elt_undefined((~ A),H,B) :- !,
   body_elt_undefined(A,H,B).
body_elt_undefined((A;_),H,B) :-
   body_elt_undefined(A,H,B).
body_elt_undefined((_;A),H,B) :- !,
   body_elt_undefined(A,H,B).
body_elt_undefined(call(A),H,B) :- !,
   body_elt_undefined(A,H,B).
body_elt_undefined(_ \= _,_,_) :- !,fail.
%body_elt_undefined(A,_,_) :-
%   askabl(A),!,fail.
%body_elt_undefined(A,_,_) :-
%   assumabl(A),!,fail.
body_elt_undefined(A,_,_) :-
   builtin(A),!,fail.
body_elt_undefined(A,_,_) :-
   hypothesis(A,_),!,fail.
body_elt_undefined(A,_,_) :-
```

```
   rul(A,_),!,fail.
body_elt_undefined(A,H,B) :-
   writeln(['Warning: no clauses for ',A,' in rule ',(H <- B),'.']),!,fail.
/*
```

5 Miscellaneous

5.1 File Handling

To consult a probabilistic Horn abduction file, you should do a,

> **thcons**(*filename).*

The following is the definition of *thcons*. Basically we just keep reading the file and executing the commands in it until we stop. This does not clear any previous database. If you reconsult a file you will get multiple instances of clauses and this will undoubtedly screw you up.

```
 */
thcons(File) :-
   current_input(OldFile),
   open(File,read,Input),
   set_input(Input),
   read(T),
   read_all(T),
   set_input(OldFile),
   writeln(['ICL theory ',File,' consulted.']).
/*


 */
read_all(end_of_file) :- !.

read_all(T) :-
   (call(T);format('Warning: ~w failed~n',[T])),
   read(T2),
   read_all(T2).
/*
```

5.2 Utility Functions

5.2.1 List Predicates

$append(X, Y, Z)$ is the normal append function

```
 */
append([],L,L).

append([H|X],Y,[H|Z]) :-
   append(X,Y,Z).
/*

 */
union([],L,L).

union([H|X],Y,Z) :-
   member(H,Y),!,
   union(X,Y,Z).
union([H|X],Y,[H|Z]) :-
   union(X,Y,Z).
/*

 */
member(A,[A|_]).
member(A,[_|R]) :-
   member(A,R).
/*

 */
memberR(A,[A|R],R).
memberR(A,[_|T],R) :-
   memberR(A,T,R).
/*

 */
subset([],_).
subset([H|T],L) :-
   member(H,L),
   subset(T,L).
/*
```

5.2.2 Term Management

```
 */
makeground(T) :-
   numbervars(T,0,_).
/*
```

$repvar(X, X1, T, T1)$ replaces each occurrence of X in T by $X1$ forming $T1$.

```
 */
repvar(X,X1,Y,X1) :- X==Y, !.
repvar(_,_,Y,Y) :- var(Y), !.
repvar(_,_,Y,Y) :- ground(Y), !.
repvar(X,X1,[H|T],[H1|T1]) :- !,
   repvar(X,X1,H,H1),
   repvar(X,X1,T,T1).
repvar(X,X1,T,T1) :-
   T =.. L,
   repvar(X,X1,L,L1),
   T1 =.. L1.
/*
```

5.2.3 Output

```
 */
writeln([]) :- nl.
writeln([H|T]) :-
   write(H),
   writeln(T).
/*
```

References

[1] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94:7–56, 1997. special issue on economic principles of multi-agent systems.

[2] D. Poole. Abducing through negation as failure: stable models in the Independent Choice Logic. *Journal of Logic Programming*, to appear, 1998.

Index